CODE PATH *ORG

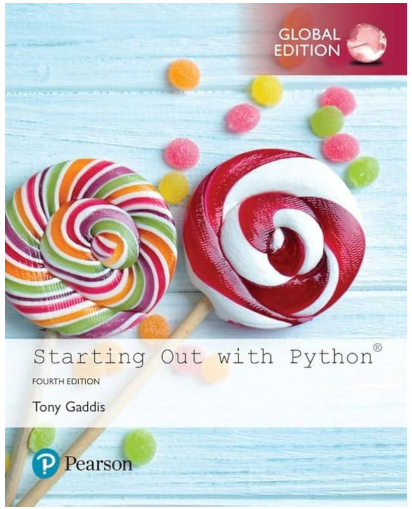
Recursion

CIIC 3015

Karelys López Rivera

Lecture Goals

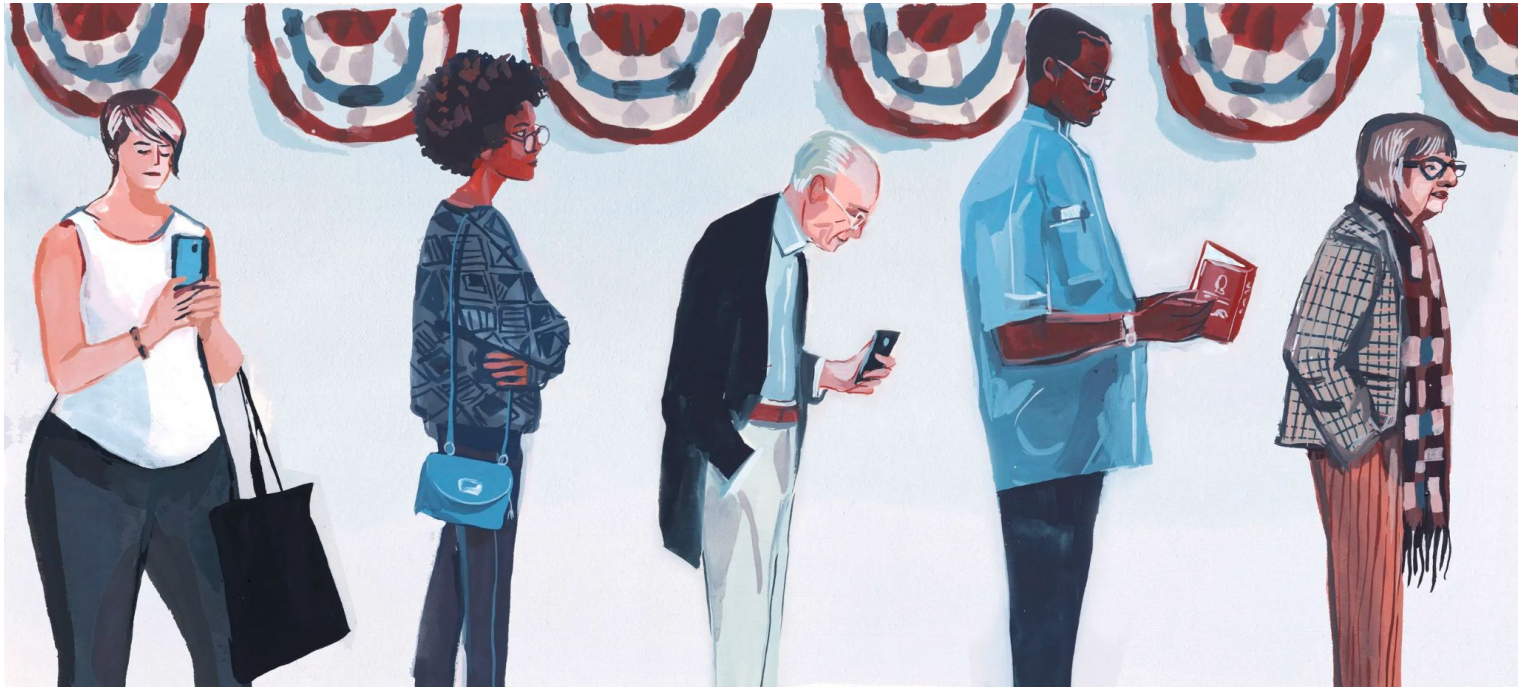
- **Recursion**
  - What are they?
  - Why do we need them?



Starting Out With Python (4th Edition) Chapter 12

Scenario

Imagine you are waiting in a line…



by: Jean Carlos Mendez

Imagine you are waiting in a line…

How do you determine how long the line is?

You could just get out of line and count!

by: Jean Carlos Mendez

What if it is a really long line?

by: Jean Carlos Mendez

Imagine you are waiting in a line...

How do you determine how long the line is?

You could still count! But it might take awhile...

It is too far to see the entire line from where you are standing

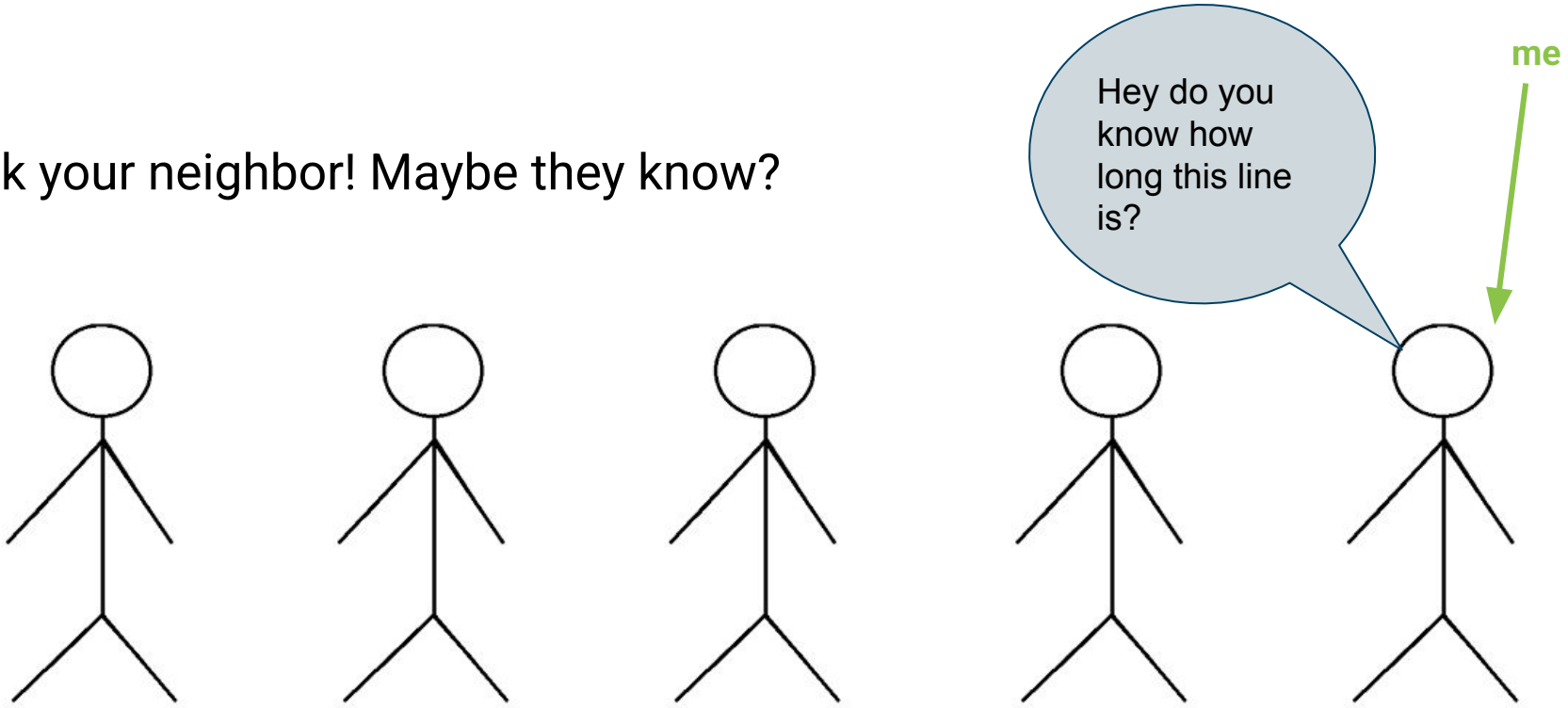
What if no one will save your spot in line? Getting out of line would make you have to go to the end :/

by: Jean Carlos Mendez

Google | GLR |

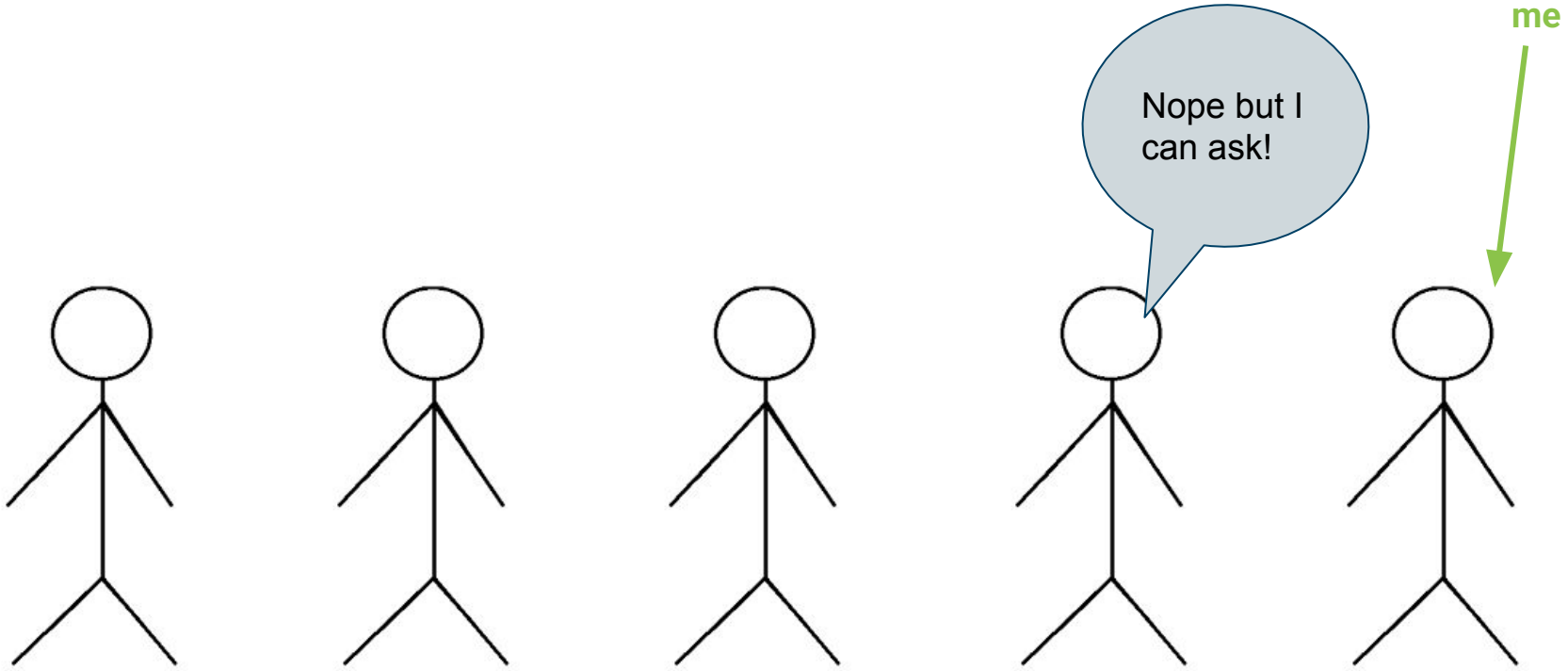
Idea!

Ask your neighbor! Maybe they know?



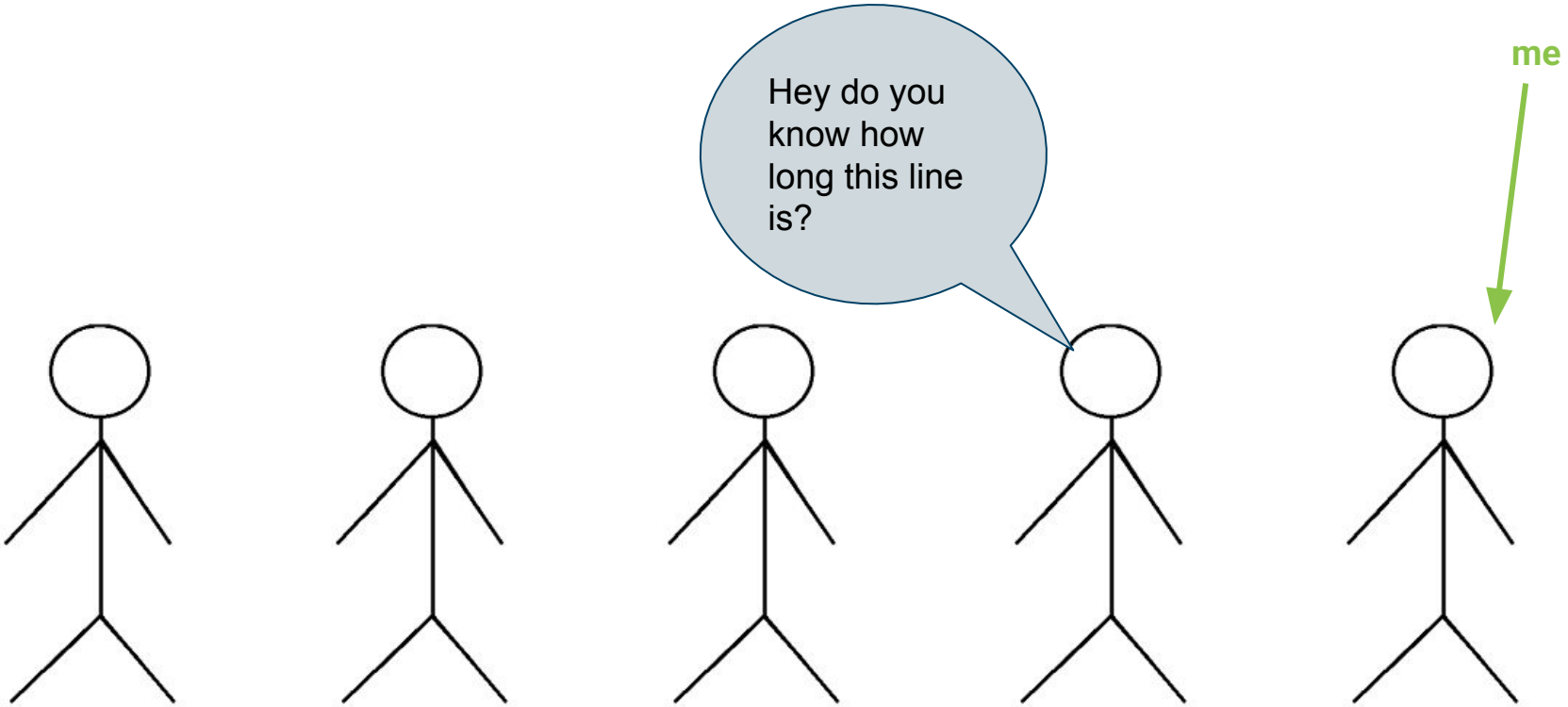
by: Jean Carlos Mendez

Idea!



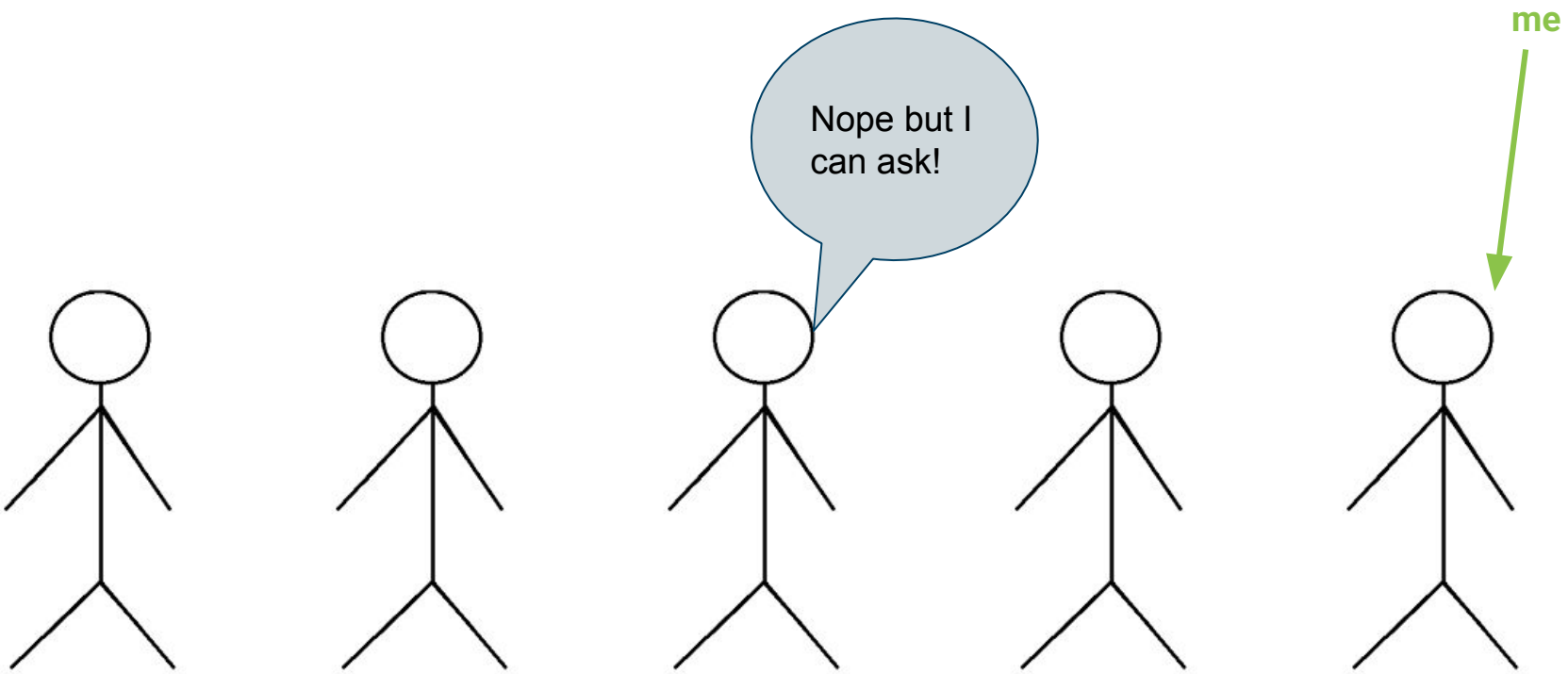
by: Jean Carlos Mendez

Idea!



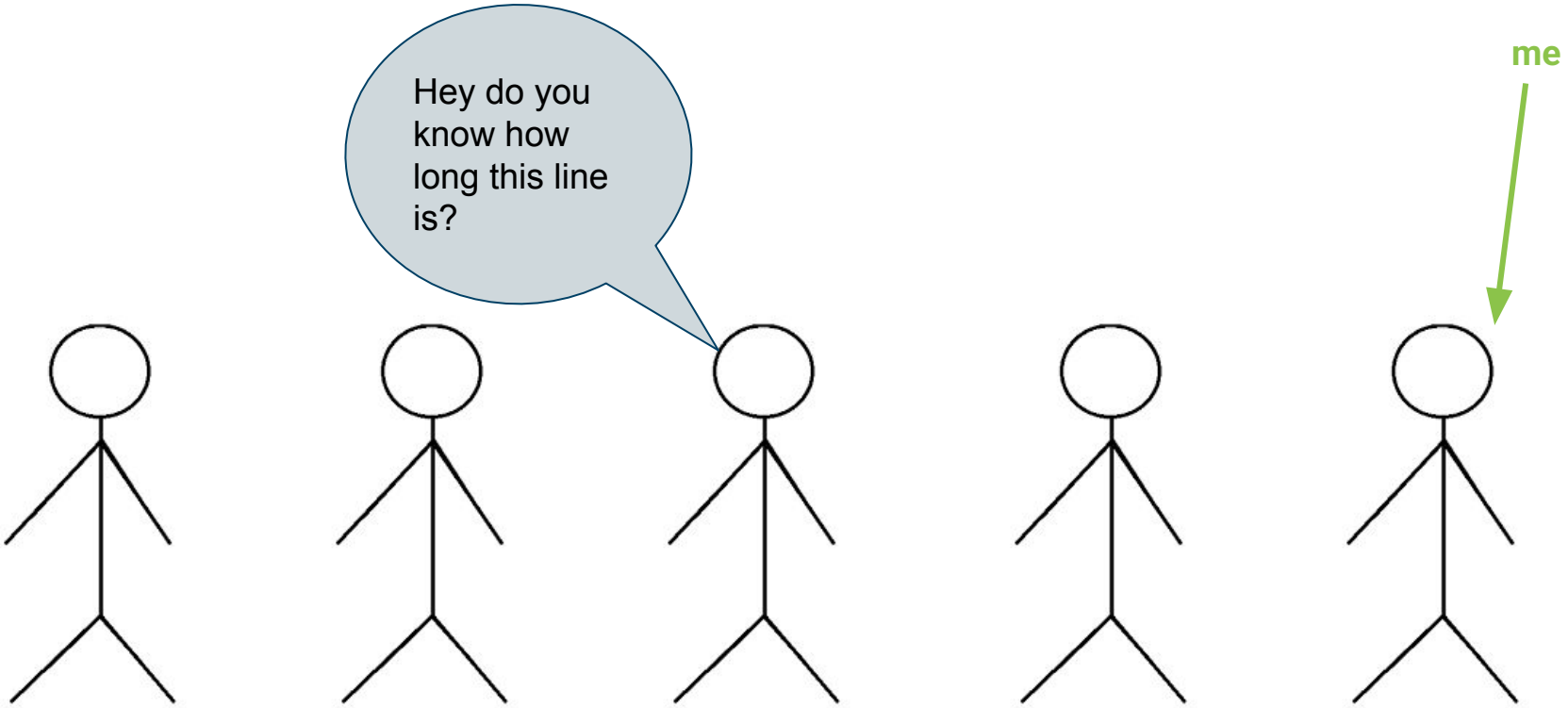
by: Jean Carlos Mendez

Idea!



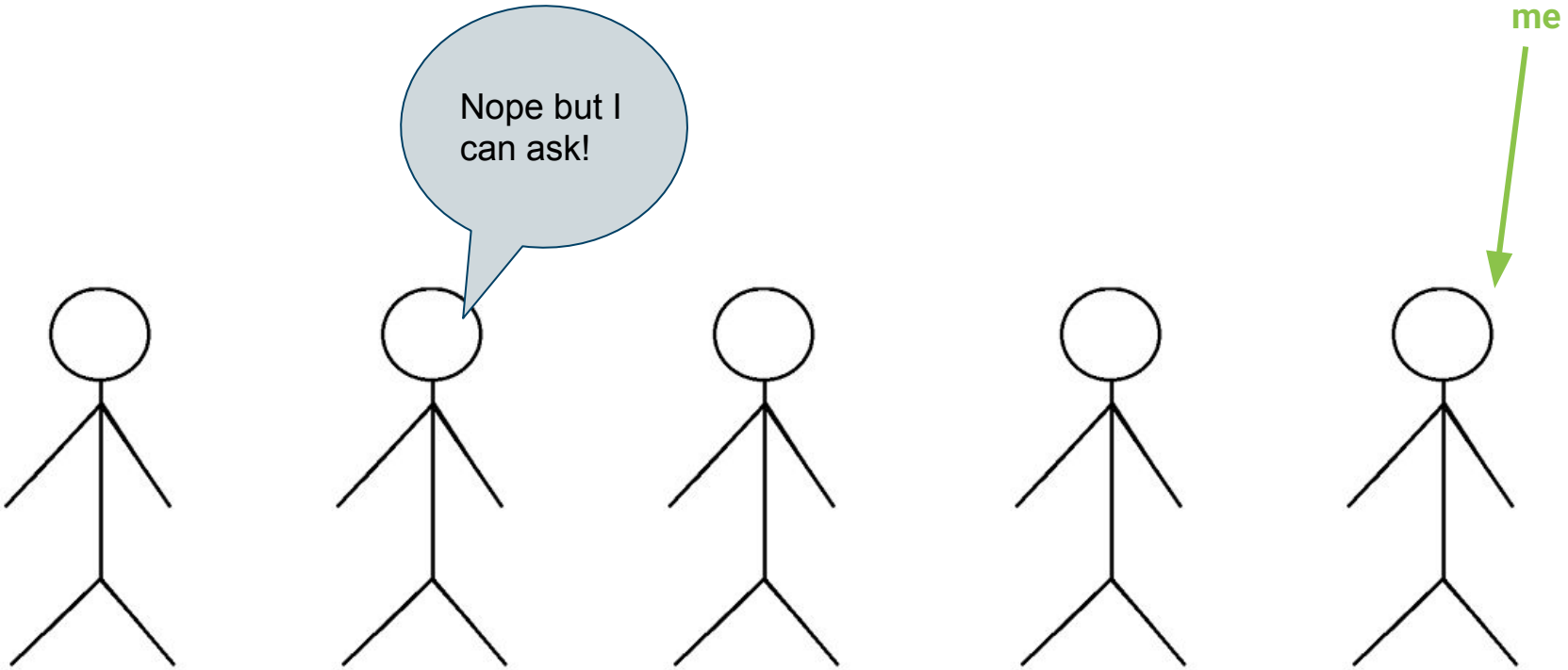
by: Jean Carlos Mendez

Idea!



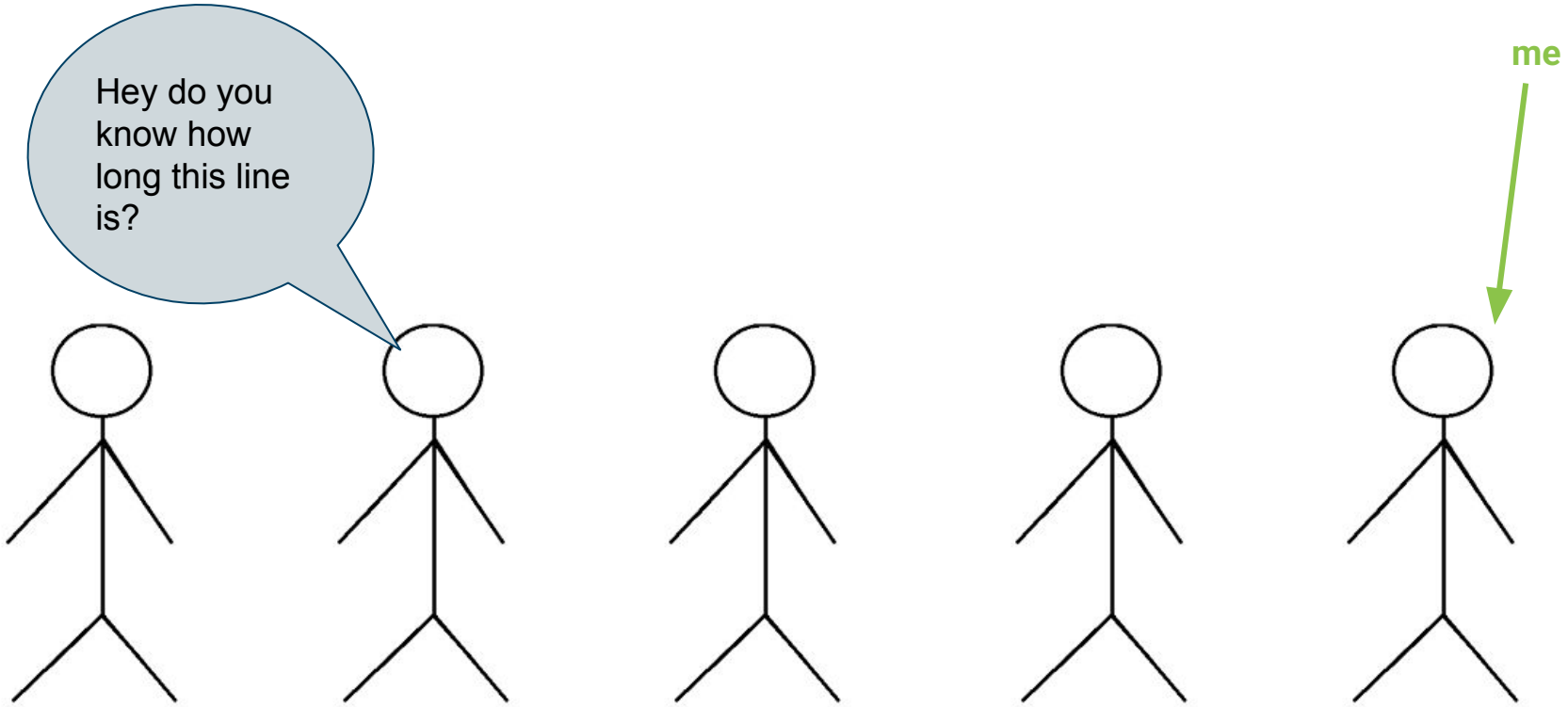
by: Jean Carlos Mendez

Idea!



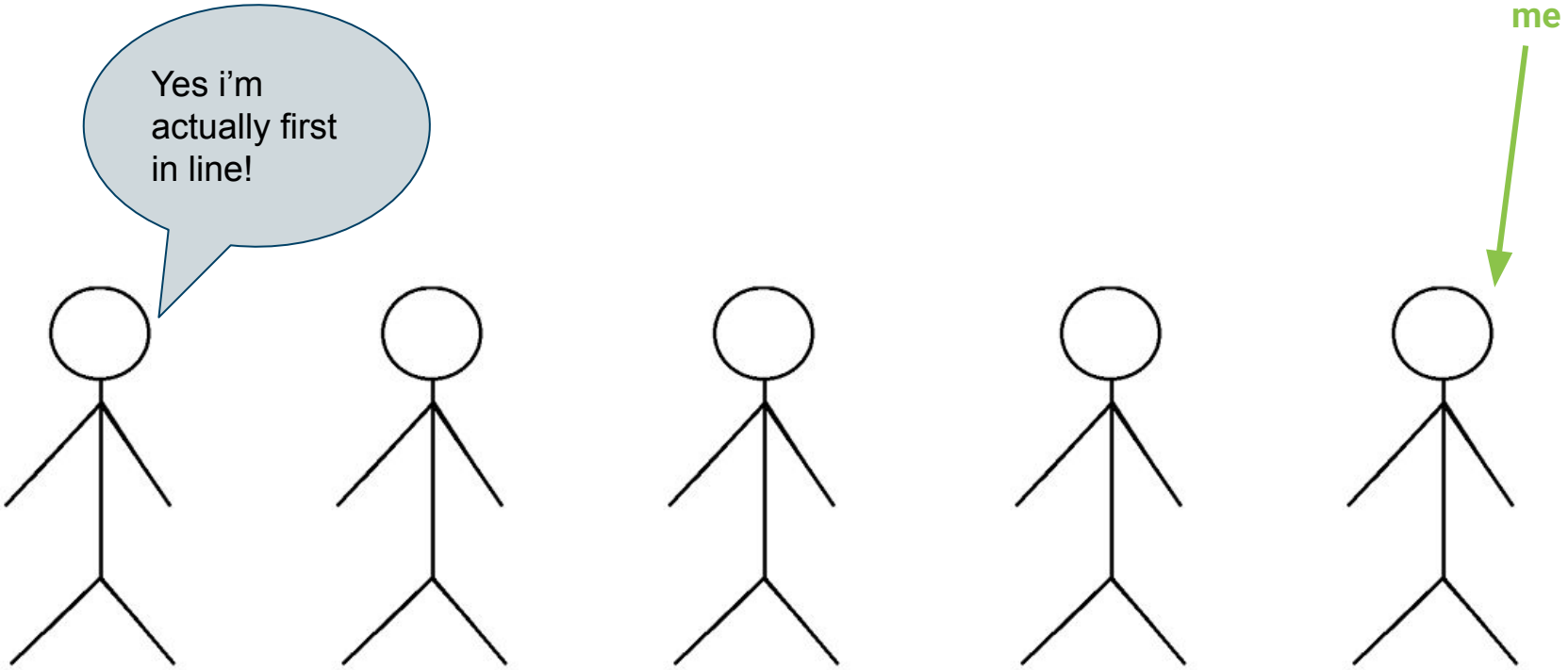
by: Jean Carlos Mendez

Idea!



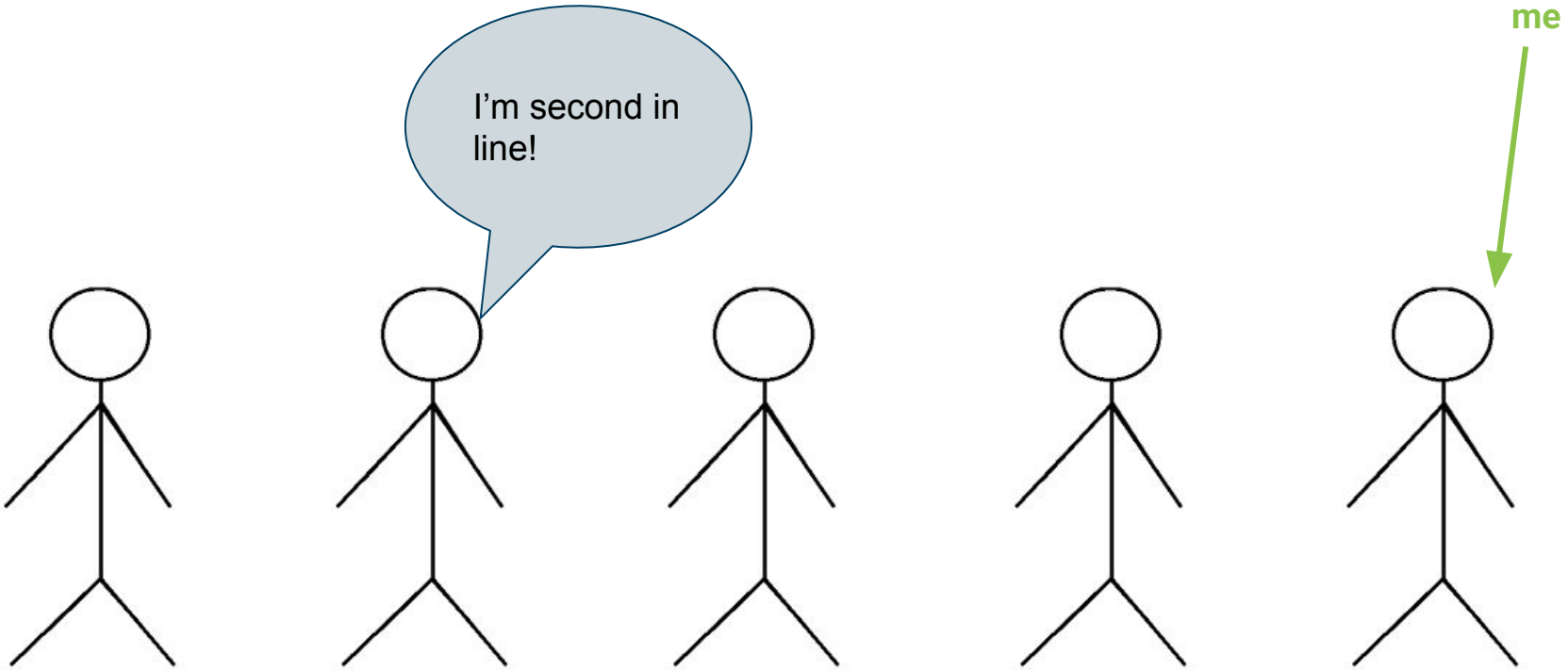
by: Jean Carlos Mendez

Idea!



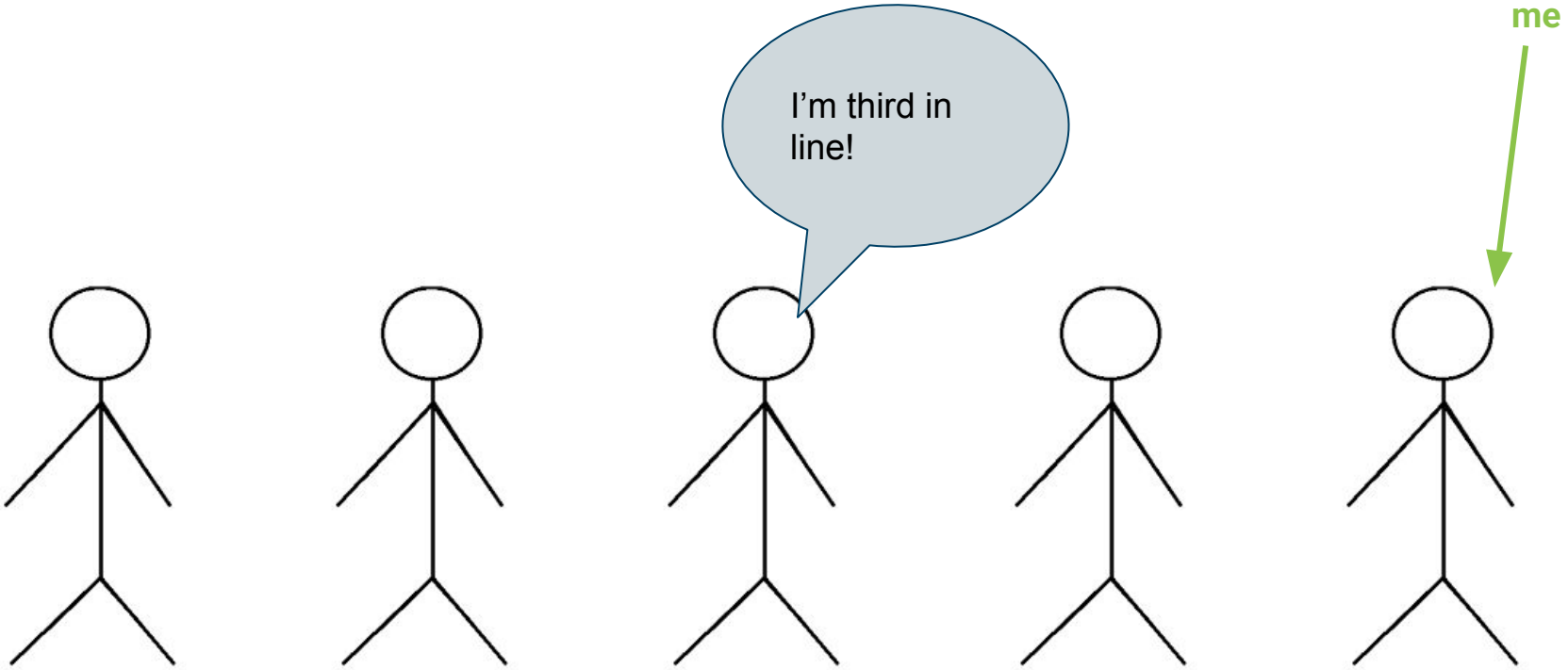
by: Jean Carlos Mendez

Idea!



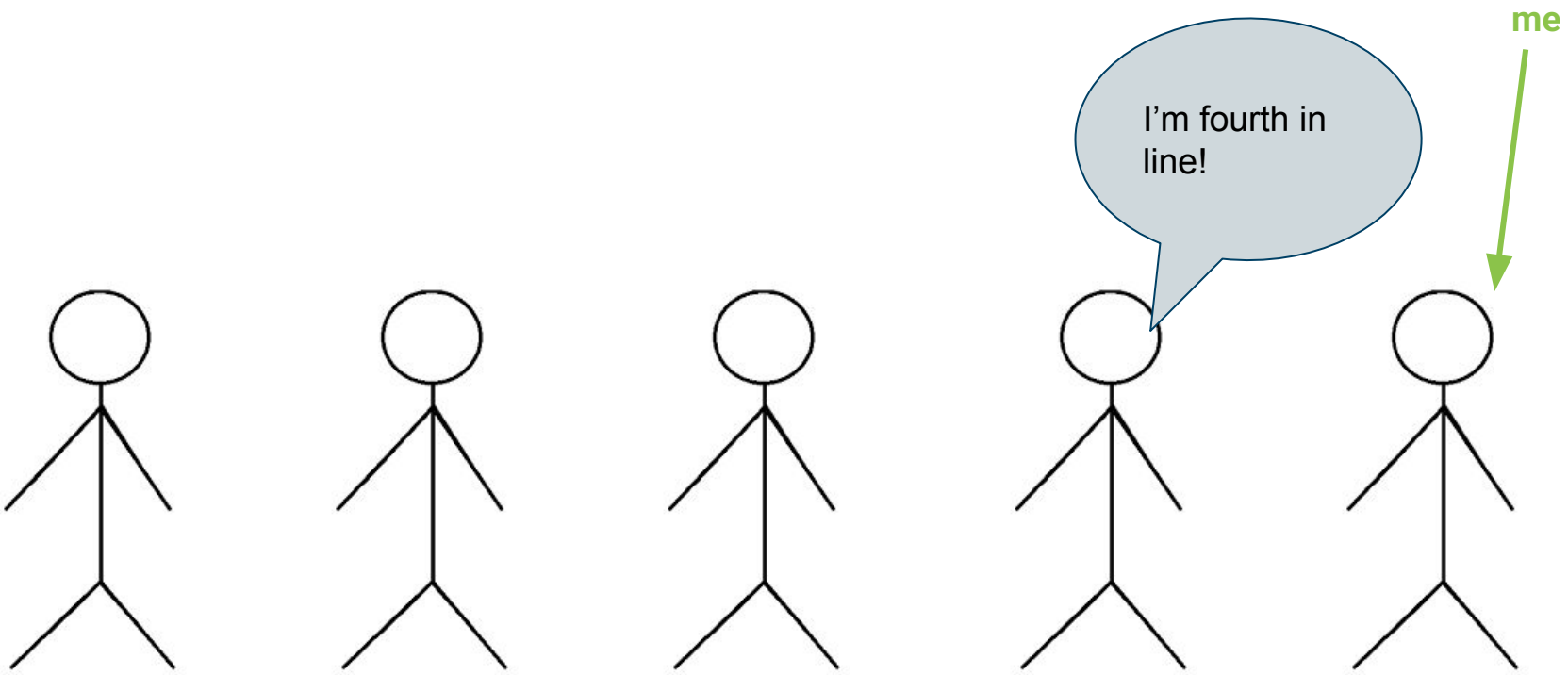
by: Jean Carlos Mendez

Idea!



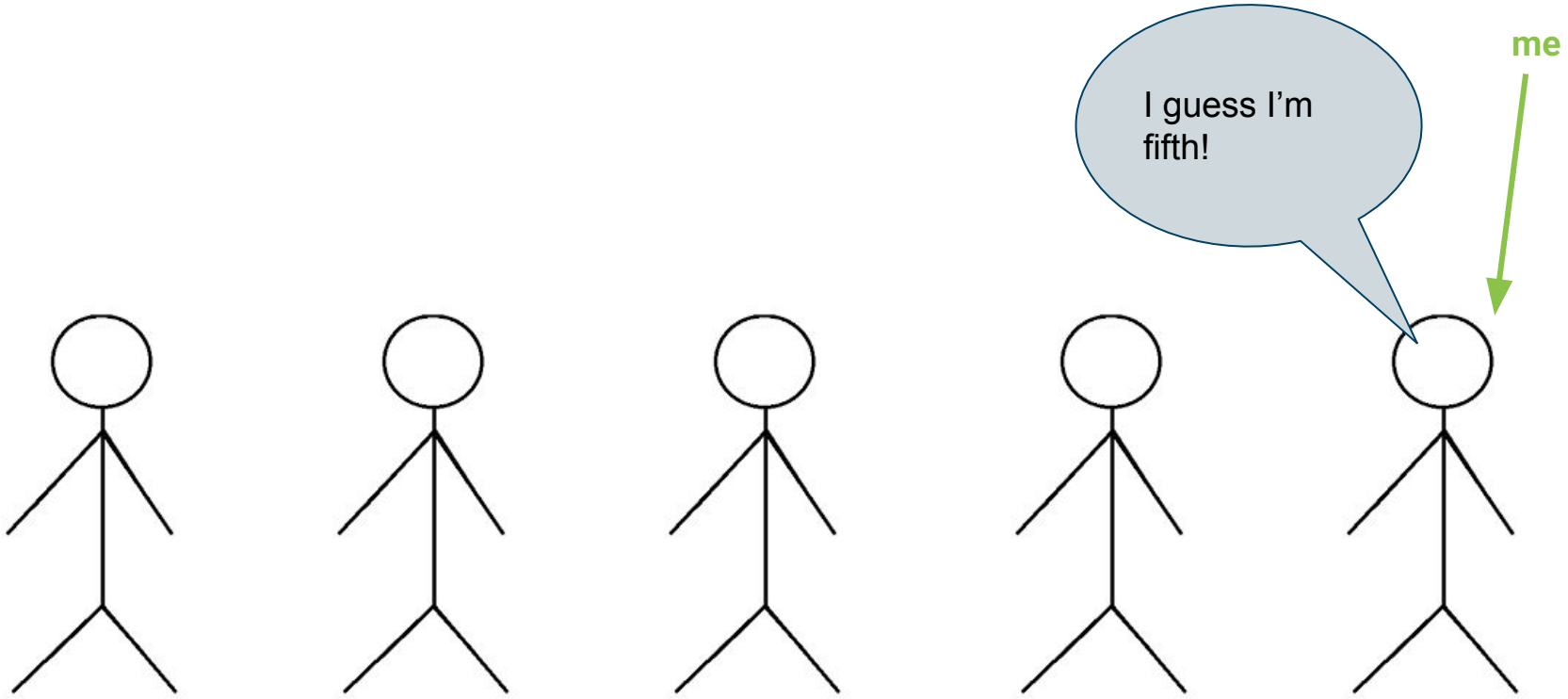
by: Jean Carlos Mendez

Idea!



by: Jean Carlos Mendez

Idea!

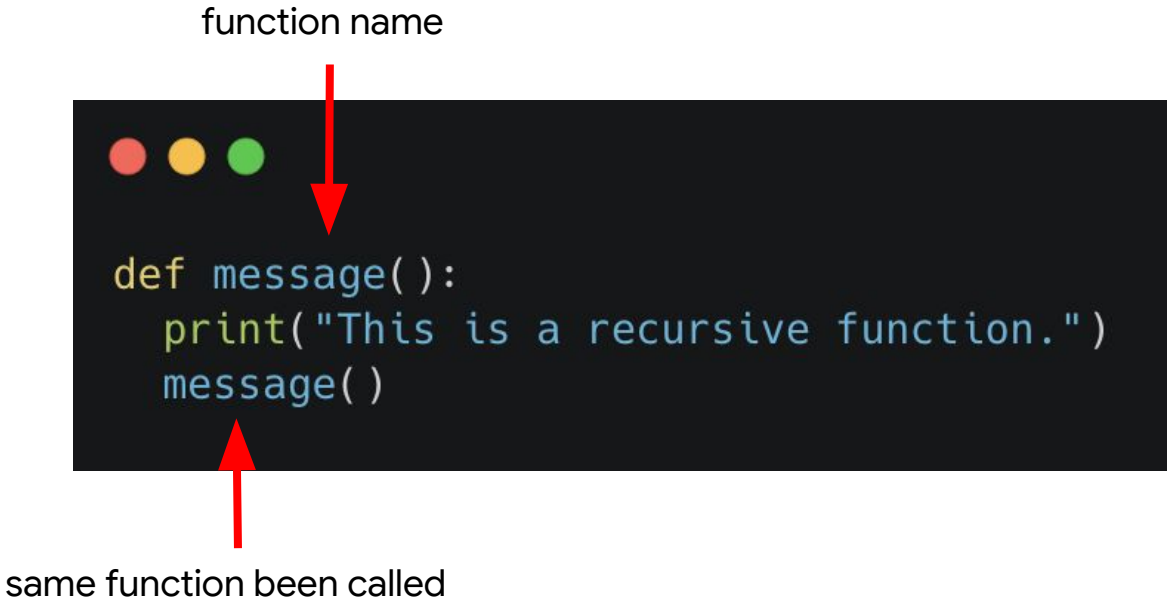


by: Jean Carlos Mendez

Recursion

Recursion

A **recursive** function is a function that calls itself



Infinite Recursion

🚨 Problem: This causes infinite recursion — no stop condition (like an infinite loop).

```
def message():
  print("This is a recursive function.")
  message()
```

A recursive function must have a stopping condition (base case) to end the repetition.

Check your understanding

What happens if a recursive function never reaches its base case?

it will cause an infinite recursive call ♾️😮

How do we avoid it?... 🤔

Problem Solving with Recursion

A problem can be solved recursively if it can be broken into smaller versions of itself.

Structure of Recursive Functions

- base case
- recursive case

Base Case

- The base case is the simplest version of a problem — one that can be solved without any further recursion.
- It's what stops the recursive calls.
- If a recursive function doesn't have a base case (or never reaches it), it will run forever and cause a stack overflow error.

```
def line_length(person):
    if (you are first in line):
        return 1
     # To be continue
```

Recursive Case

- The recursive case is the part of the function that calls itself on a smaller or simpler version of the same problem.
- It's how the problem gets "broken down" step by step.

```
def line_length(person):
    if (you are first in line):
        return 1
    else:
        # ask neighbor what position they are in and wait
        # for their response. Add 1
        return line_length(neighbor) + 1
```

Base Case vs Recursive Case

Base Case	Recursive Case
When to stop	How to keep going
Simple direct solution	Calls itself with smaller input

Check your understanding

The function stops calling itself when _____.

The base case condition becomes false.

Let’s work together: Factorial

Product of all positive integers less than or equal to N, starting from the factorial of 0 equaling 1.

Mathematical definition:

- $n! = 1 \times 2 \times 3 \times \ldots \times n$

Example: Factorial(5):

- 5! = 1 * 1 * 2 * 3 * 4 * 5 = 120

Let’s work together: Factorial

For our problem:

Function name is going to be factorial(). The input and output is a number

Examples:

factorial(0) -> 1

factorial(1) -> 1

factorial(2) -> 2

factorial(3) -> 6

factorial(4) -> 24

factorial(5) -> 120

Reminder: Factorial Using Loops

```
def factorial(n):
    fact = 1
    for num in range(1, n+1):
        fact *= num
    return fact
```

Thinking Recursively: Factorial

Can I define this problem in terms of itself?

Hint: what if I start from the top instead of from the bottom?

factorial(5) = factorial(4) * 5

factorial(4) = factorial(3) * 4

factorial(3) = factorial(2) * 3

factorial(2) = factorial(1) * 2

factorial(1) = factorial(0) * 1

Base case! ➡️ factorial(0) = 1

Thinking Recursively: Factorial

Can I define this problem in terms of itself?

Hint: what if I start from the top instead of from the bottom?

factorial(5) = factorial(4) * 5

factorial(4) = factorial(3) * 4

factorial(3) = factorial(2) * 3

factorial(2) = factorial(1) * 2

factorial(1) = factorial(0) * 1

factorial(0) = 1

Thinking Recursively: Factorial

Can I define this problem in terms of itself?

Hint: what if I start from the top instead of from the bottom?

factorial(5) = factorial(4) * 5

factorial(4) = factorial(3) * 4

factorial(3) = factorial(2) * 3

factorial(2) = factorial(1) * 2

factorial(1) = 1 * 1 = 1

Thinking Recursively: Factorial

Can I define this problem in terms of itself?

Hint: what if I start from the top instead of from the bottom?

factorial(5) = factorial(4) * 5

factorial(4) = factorial(3) * 4

factorial(3) = factorial(2) * 3

factorial(2) = 1 * 2 = 2

Thinking Recursively: Factorial

Can I define this problem in terms of itself?

Hint: what if I start from the top instead of from the bottom?

factorial(5) = factorial(4) * 5

factorial(4) = factorial(3) * 4

factorial(3) = 2 * 3 = 6

Thinking Recursively: Factorial

Can I define this problem in terms of itself?

Hint: what if I start from the top instead of from the bottom?

factorial(5) = factorial(4) * 5

factorial(4) = 6 * 4 = 24

Thinking Recursively: Factorial

Can I define this problem in terms of itself?

Hint: what if I start from the top instead of from the bottom?

factorial(5) = 24 * 5

Thinking Recursively: Factorial

Can I define this problem in terms of itself?

Hint: what if I start from the top instead of from the bottom?

factorial(5) = 120

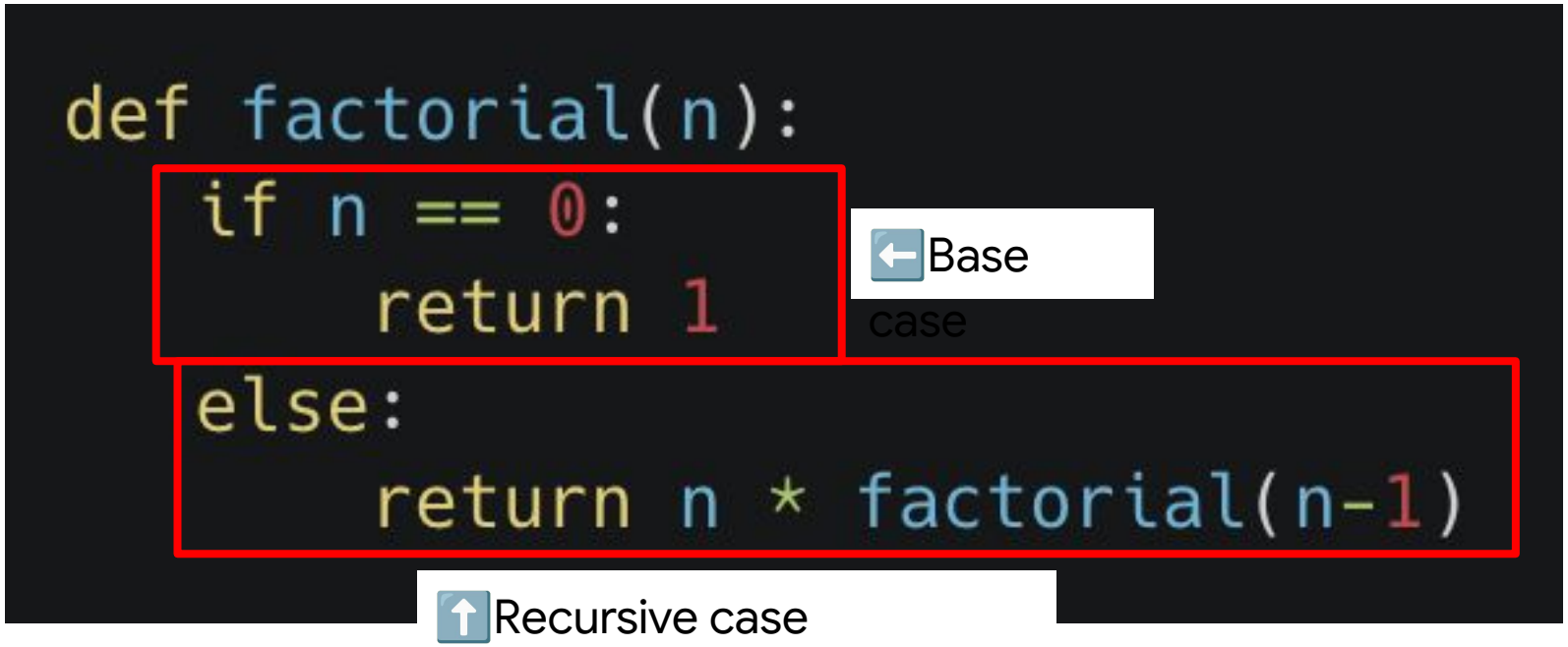
We got to result! 🥳 But how?🤔

Thinking Recursively: Factorial

- Instead of starting from bottom like we did iteratively (using loop) we started from the top
- Our base case from we saw and know about factorial is when the input is 0 the number is 1
- The recursive case is taking the input and multiplying it by factorial(n-1)

```
def factorial(n):
    if n == 0:
        return 1
    else:
        return n * factorial(n-1)
```

Thinking Recursively: Factorial



Thinking Recursively: Factorial in Python Tutor

Let’s work together: Fibonacci

Special sequence of numbers where a number is the sum of last two numbers. The 0th Fibonacci number is 0. The 1st Fibonacci number is 1. Every number after that is the sum of the two previous numbers.

Mathematical definition:

- n = 0, 1, fibonacci(n-1)+fibonacci(n-2), ...

Example: Fibonacci(5):

- 5 = 0, 1, 1, 2, 3, 5

Let’s work together: Fibonacci

Example: Fibonacci(5):

- 5 = 0, 1, 1, 2, 3, 5

Number	Value in Fibonacci (Fib is short for Fibonacci)
0	0
1	1
2	Fib(2-1)+Fib(2-2) ➡️ Fib(1)+Fib(0) ➡️1+0 ➡️ 1
3	Fib(3-1)+Fib(3-2) ➡️ Fib(2)+Fib(1) ➡️1+1 ➡️ 2
4	Fib(4-1)+Fib(4-2) ➡️ Fib(3)+Fib(2) ➡️2+1➡️ 3
5	Fib(5-1)+Fib(5-2) ➡️ Fib(4)+Fib(3) ➡️3+1 ➡️ 5

Let’s work together: Factorial

For our problem:

Function name is going to be fibonacci(). The input and output is a number

Examples:

fibonacci(0) -> 0

fibonacci(1) -> 1

fibonacci(2) -> 1

fibonacci(3) -> 2

fibonacci(4) -> 3

fibonacci(5) -> 5

Reminder: Fibonacci Using Loops

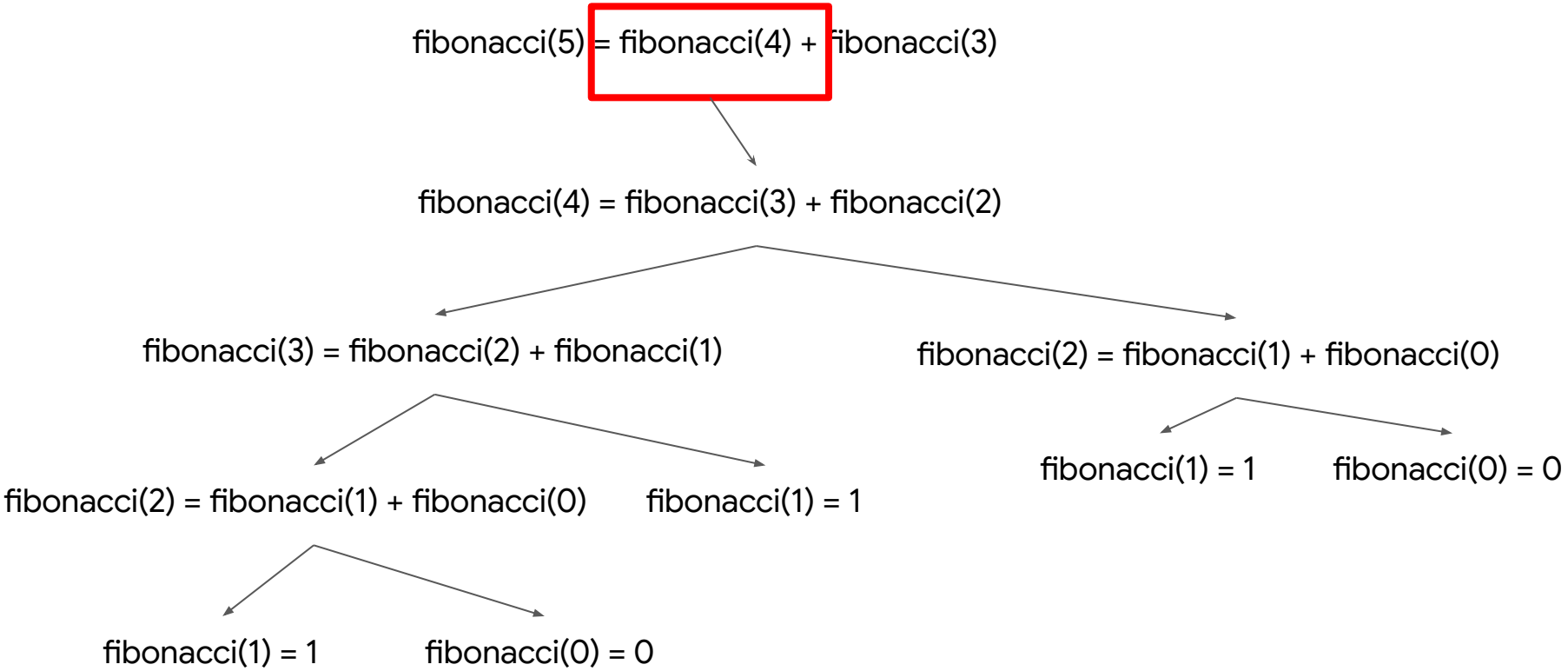
```
def fibonacci(num):
    if num <= 1:
        return num
    num_1 = 0
    num_2 = 1
    for _ in range(num - 1):
        temp = num_2
        num_2 = num_2 + num_1
        num_1 = temp
    return num_2
```

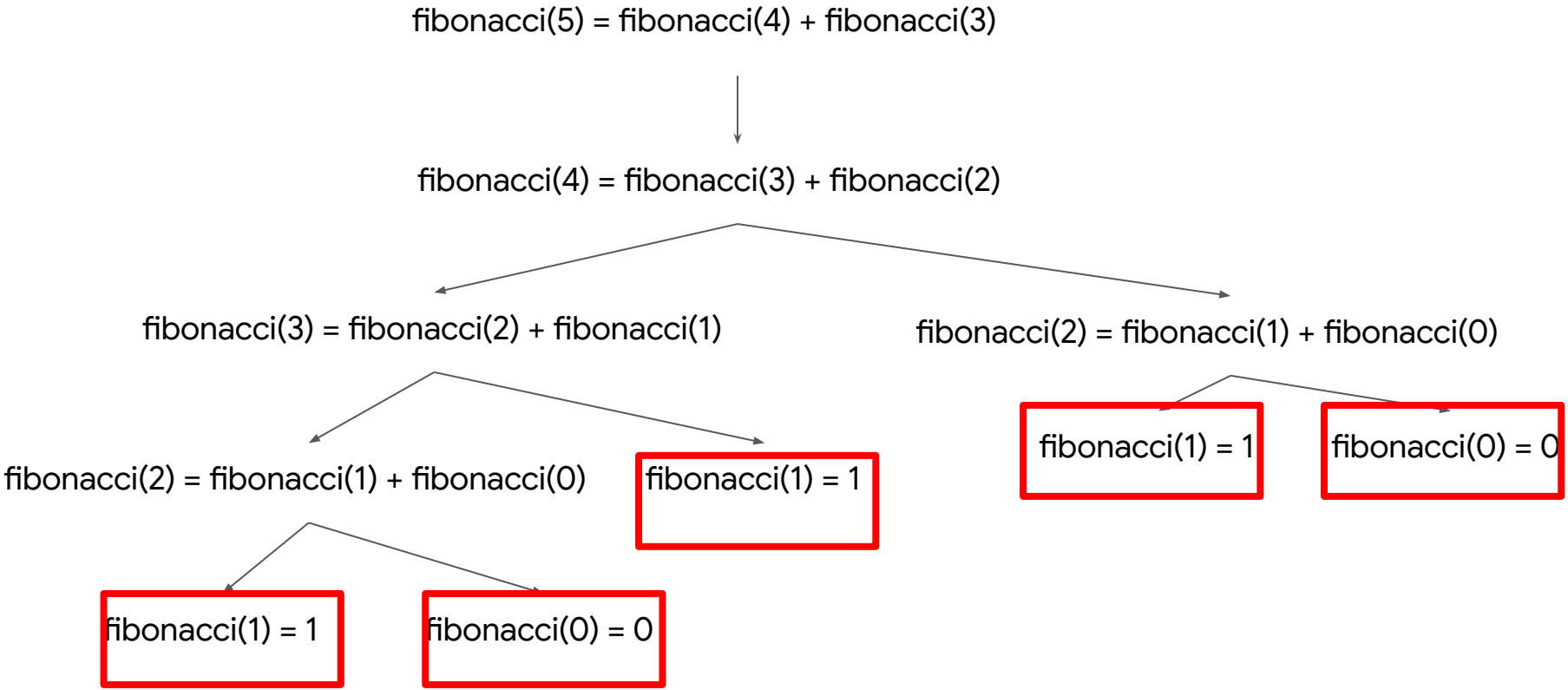
Thinking Recursively: Fibonacci

Can I define this problem in terms of itself?

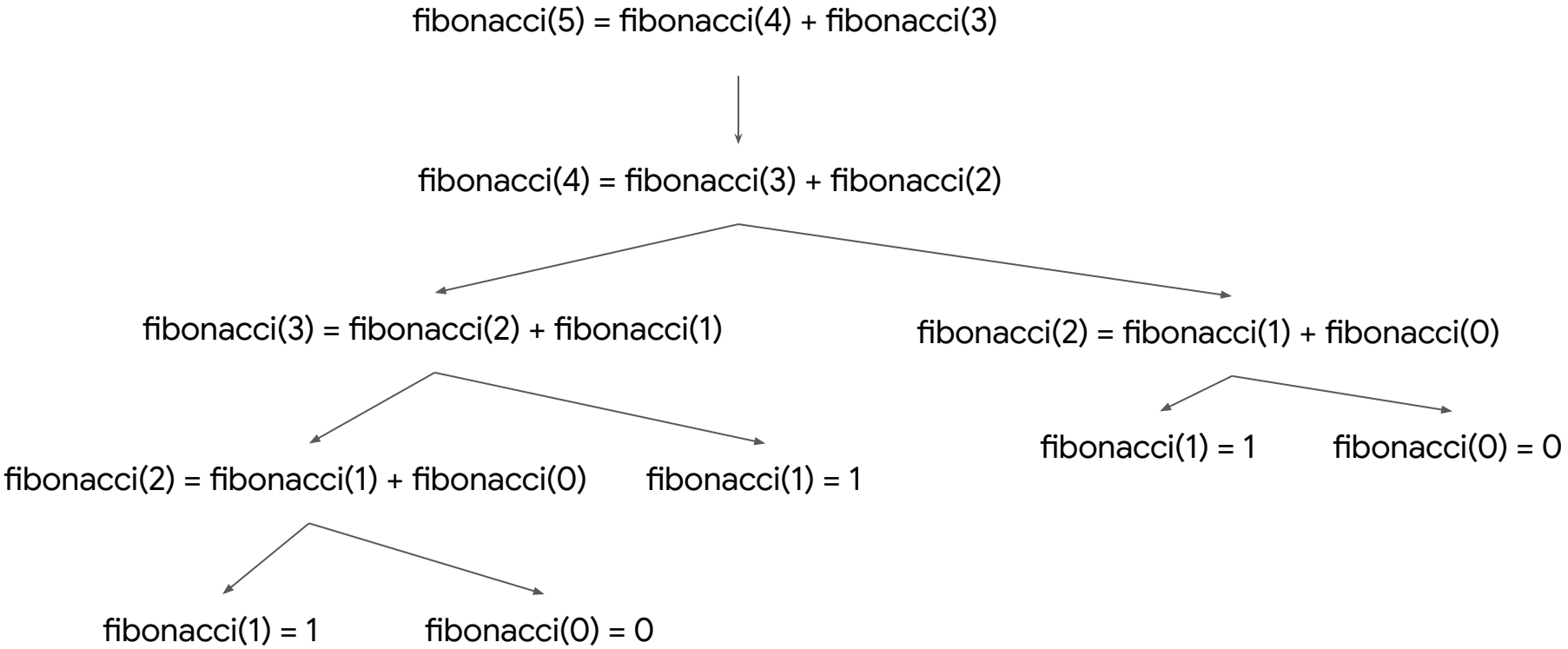
Hint: what if I start from the top instead of from the bottom?

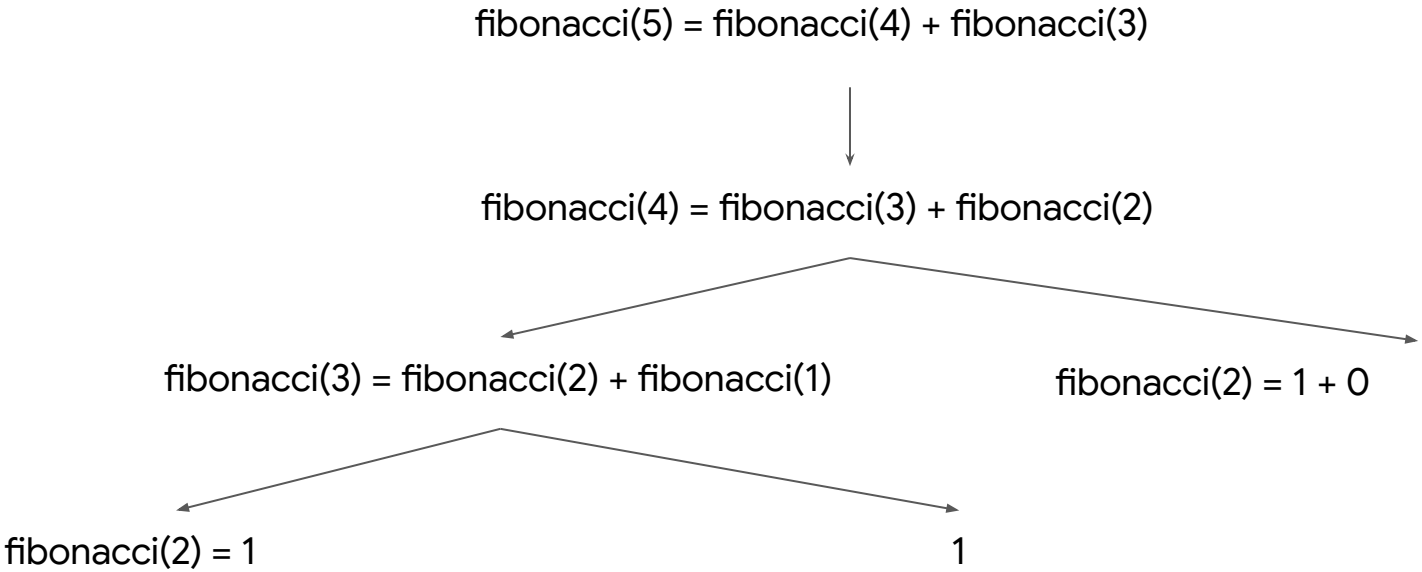
fibonacci(5) = fibonacci(4) + fibonacci(3)

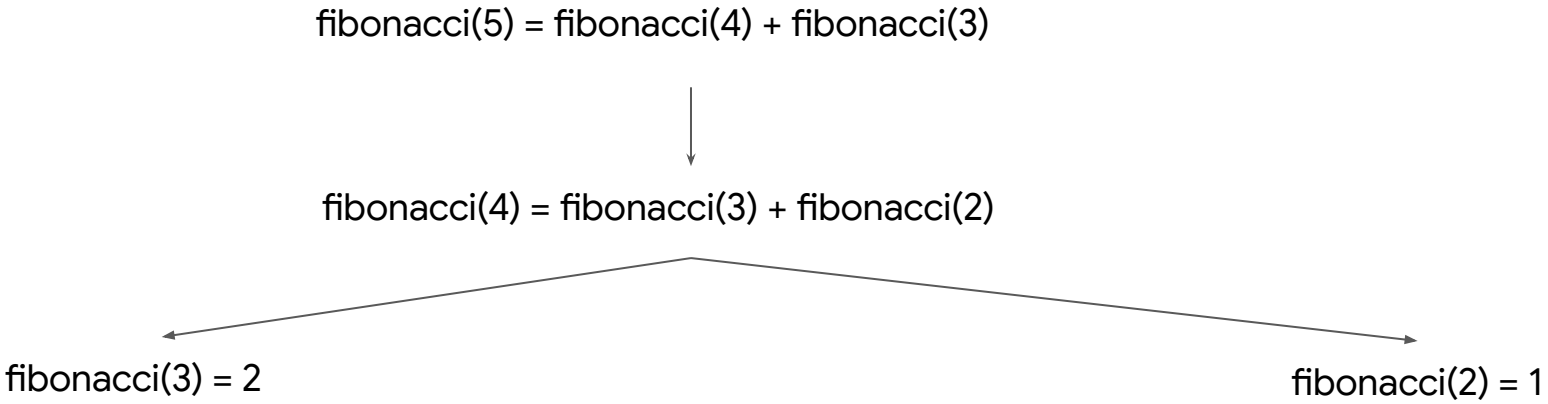

fibonacci(5) = fibonacci(4) + fibonacci(3)
fibonacci(4) = fibonacci(3) + fibonacci(2)
fibonacci(3) = fibonacci(2) + fibonacci(1)
fibonacci(2) = fibonacci(1) + fibonacci(0)
fibonacci(1) = 1
fibonacci(0) = 0
fibonacci(2) = fibonacci(1) + fibonacci(0)
fibonacci(1) = 1
fibonacci(1) = 1
fibonacci(0) = 0


fibonacci(5) = fibonacci(4) + fibonacci(3)
fibonacci(4) = fibonacci(3) + fibonacci(2)
fibonacci(3) = fibonacci(2) + fibonacci(1)
fibonacci(2) = fibonacci(1) + fibonacci(0)
fibonacci(1) = 1
fibonacci(1) = 1
fibonacci(0) = 0
fibonacci(2) = fibonacci(1) + fibonacci(0)
fibonacci(1) = 1
fibonacci(0) = 0

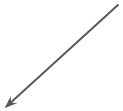
The known values are base cases!


fibonacci(5) = fibonacci(4) + fibonacci(3)
fibonacci(4) = fibonacci(3) + fibonacci(2)
fibonacci(3) = fibonacci(2) + fibonacci(1)
fibonacci(2) = fibonacci(1) + fibonacci(0)
fibonacci(2) = fibonacci(1) + fibonacci(0)
fibonacci(1) = 1
fibonacci(1) = 1
fibonacci(0) = 0
fibonacci(1) = 1
fibonacci(0) = 0

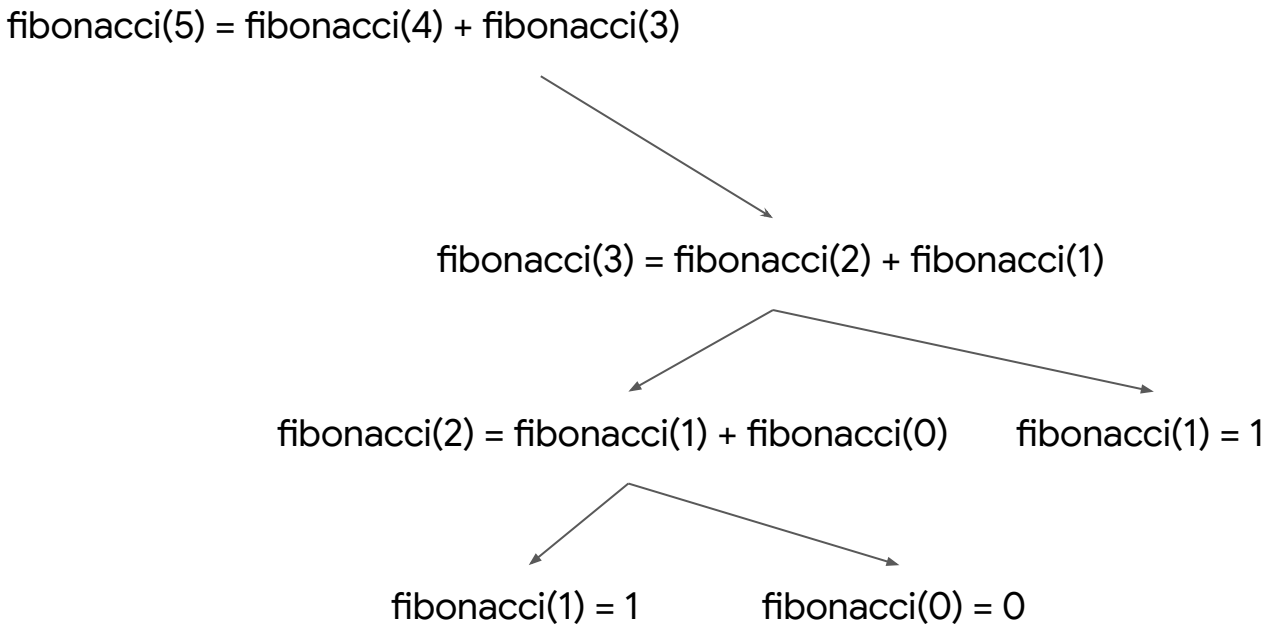

fibonacci(5) = fibonacci(4) + fibonacci(3)
fibonacci(4) = fibonacci(3) + fibonacci(2)
fibonacci(3) = fibonacci(2) + fibonacci(1)
fibonacci(2) = 1 + 0
fibonacci(2) = 1
1

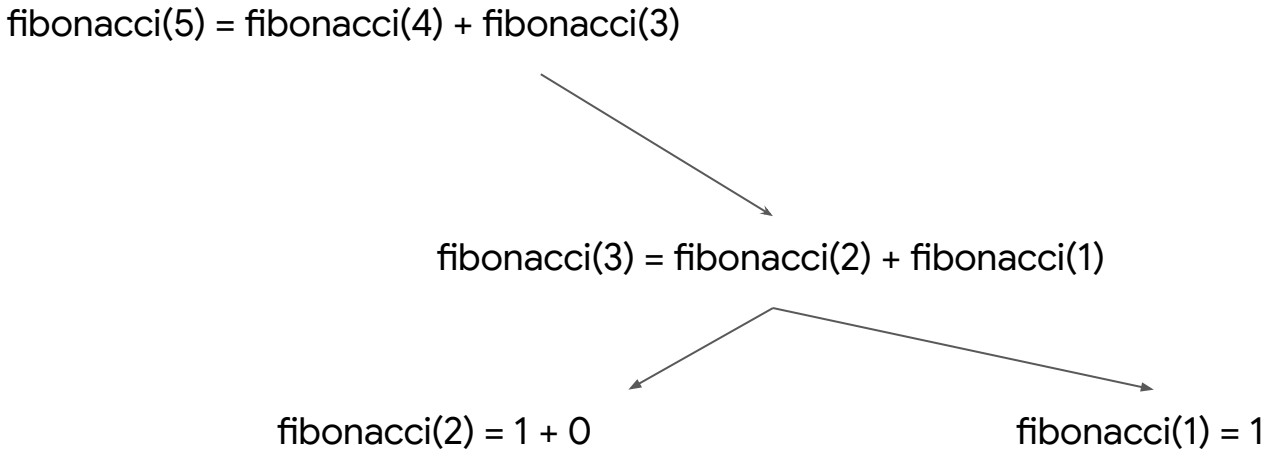

fibonacci(5) = fibonacci(4) + fibonacci(3)
fibonacci(4) = fibonacci(3) + fibonacci(2)
fibonacci(3) = 2
fibonacci(2) = 1

fibonacci(5) = fibonacci(4) + fibonacci(3)



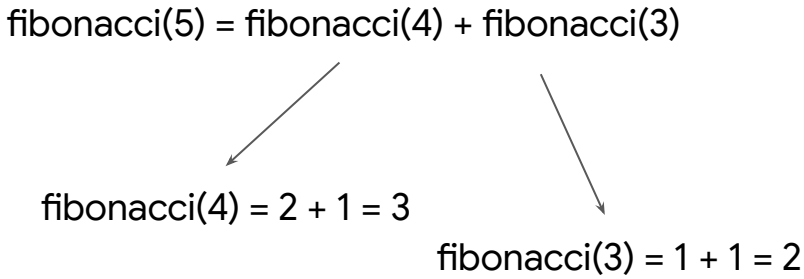
fibonacci(4) = 2 + 1 = 3


fibonacci(5) = fibonacci(4) + fibonacci(3)
fibonacci(3) = fibonacci(2) + fibonacci(1)
fibonacci(2) = fibonacci(1) + fibonacci(0)
fibonacci(1) = 1
fibonacci(1) = 1
fibonacci(0) = 0


fibonacci(5) = fibonacci(4) + fibonacci(3)
fibonacci(3) = fibonacci(2) + fibonacci(1)
fibonacci(2) = 1 + 0
fibonacci(1) = 1

fibonacci(5) = fibonacci(4) + fibonacci(3)

fibonacci(3) = 1 + 1


fibonacci(5) = fibonacci(4) + fibonacci(3)
fibonacci(4) = 2 + 1 = 3
fibonacci(3) = 1 + 1 = 2

Thinking Recursively: Fibonacci

Can I define this problem in terms of itself?

Hint: what if I start from the top instead of from the bottom?

fibonacci(5) = 5

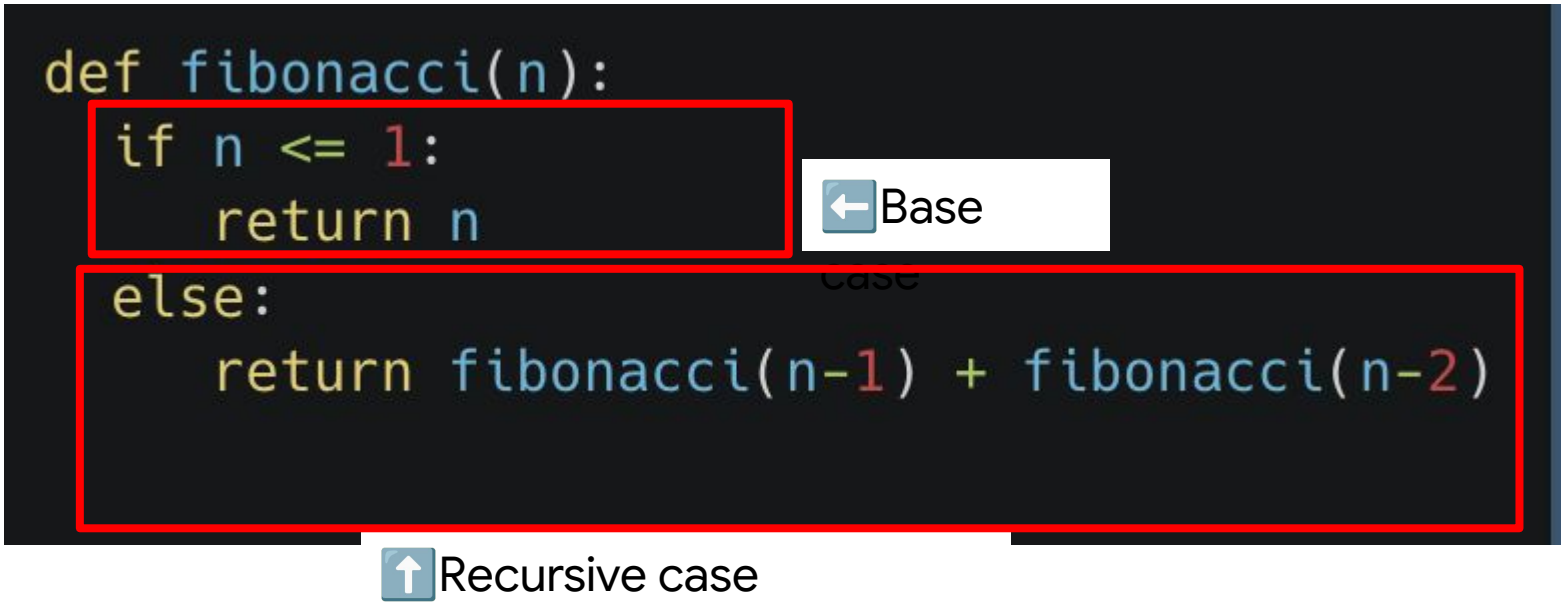
We got to result! 🥳 But how?🤔

Thinking Recursively: Fibonacci

- Instead of starting from bottom like we did iteratively (using loop) we started from the top
- Our base case from we saw and know about fibonacci is the known numbers we have 0 and 1
- The recursive case is taking the input and subtracting one and adding f(n-1)+f(n-2)

```
def fibonacci(n):
  if n <= 1:
     return n
  else:
     return fibonacci(n-1) + fibonacci(n-2)
```

Thinking Recursively: Fibonacci



Thinking Recursively: Fibonacci in Recursion Tree

Why learn recursion?

- A challenge! A different way of thinking of problems
- Can solve some kinds of problems better than iteration
- Leads to elegant, simplistic, short code (when used well)
- Some programming languages ("functional" languages such as Scheme, ML, and Haskell) use recursion exclusively (no loops)

by: Jean Carlos Mendez

IT'S CODING TIME

- sum of natural numbers
- power function
- reverse string
- is palindrome
- range sum

Questions?

Post to our Discord server

or

email:

upr-ciic3015-staff@upr.edu

karelys.lopez1@upr.edu

Give us anonymous feedback!

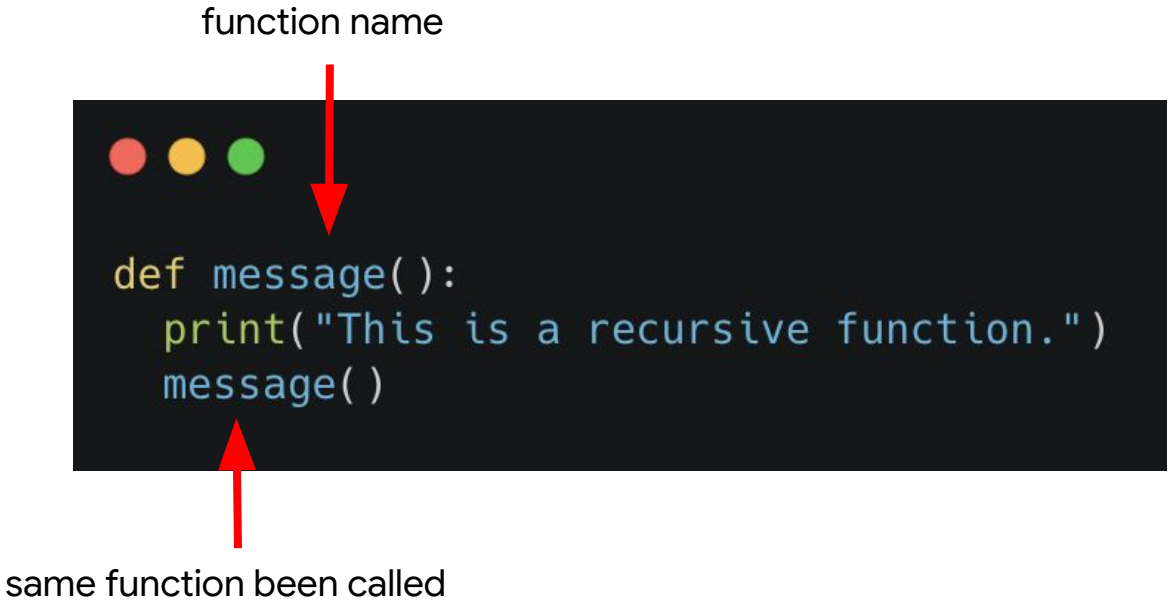
https://bit.ly/CIIC3015_F25_Feedback

CODE PATH .ORG

Recap

Recursion

A **recursive** function is a function that calls itself



Base Case vs Recursive Case

Base Case	Recursive Case
When to stop	How to keep going
Simple direct solution	Calls itself with smaller input

More Recursion Examples

Let’s work together: Reverse String

Reverses the order of characters in a string.

For example, if the input is "hello", the output should be "olleh".

Let’s work together: Reverse String

For our problem:

Function name is going to be called reverse(). The input and output is a string

Examples:

reverse(“hello”) -> “olleh”

reverse(“abc”) -> “cba”

reverse(“kayak”) -> “kayak”

reverse(“python”) -> “nohtyp”

reverse(“a”) -> “a”

Reminder: Reverse String Using Loops

```
def reverse_string(s):
    reversed_s = ""                  # Create a new empty string
    for ch in s:                     # Go through each character
        reversed_s = ch + reversed_s  # Add it to the front
    return reversed_s
```

Thinking Recursively: Reverse String

Can I define this problem in terms of itself?

reverse(“cat”) -> reverse(“at”) + “c”

reverse(“at”) -> reverse(“t”) + “a”

Base case! ➡️ reverse(“t”) -> “t”

Thinking Recursively: Reverse String

Can I define this problem in terms of itself?

reverse(“cat”) -> reverse(“at”) + “c”

reverse(“at”) -> “t” + “a” -> “ta”

Thinking Recursively: Reverse String

Can I define this problem in terms of itself?

reverse(“cat”) -> “ta” + “c” -> “tac”

We got to result! 🥳 But how?🤔

Thinking Recursively: Reverse String

- Instead of building a new string step by step like we do iteratively (using a loop),
- in recursion, each function call returns a small piece of the answer, and those pieces are automatically combined when the recursion unwinds.
- Our base case is when the string has 0 or 1 character — in that case, it's already reversed.
- The recursive case takes the first character, removes it, and calls the same function to reverse the rest (s[1:]), then adds that first character to the end.

```
def reverse(s):
    if len(s) <= 1:
        return s
    return reverse(s[1:]) + s[0]
```

Why No “New String” Variable Is Needed: Reverse String

When using a loop:

```
def reverse_string(s):
    reversed_s = ""                    # Create a new empty string
    for ch in s:                       # Go through each character
        reversed_s = ch + reversed_s   # Add it to the front
    return reversed_s
```

You need a variable (reversed_s) to store your partial result because the loop itself doesn’t remember previous steps — it only knows the current one.

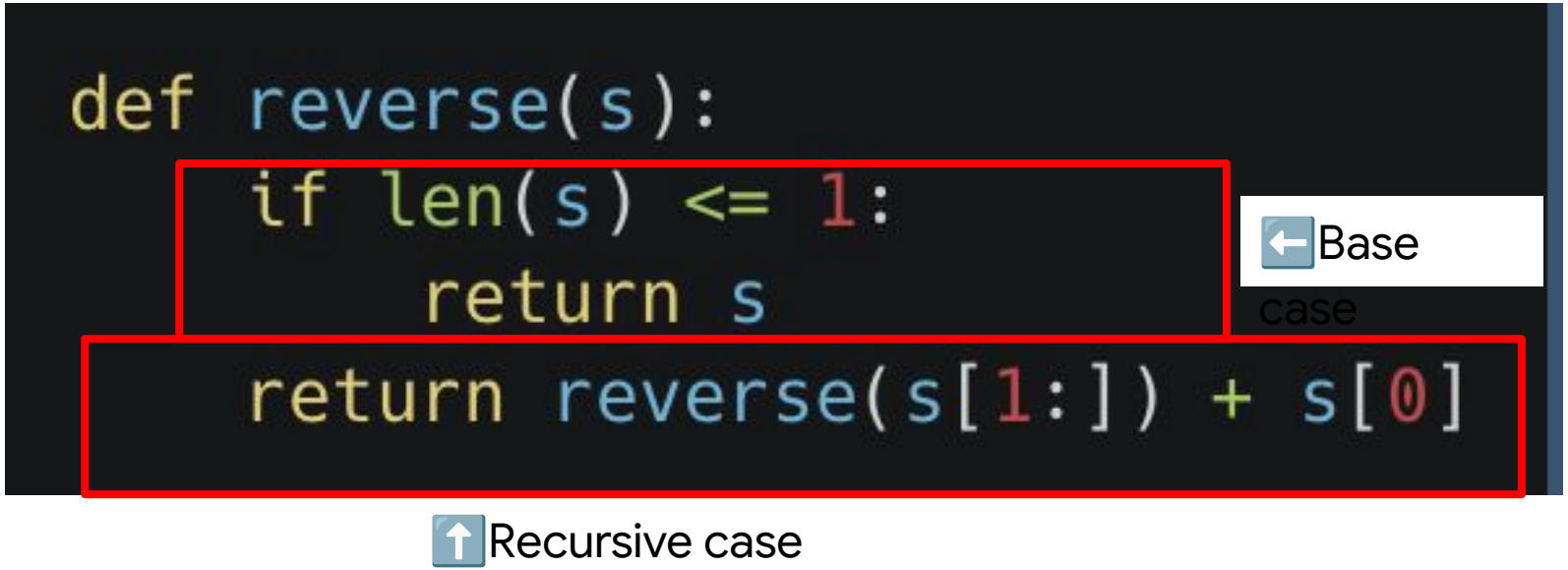
Why No “New String” Variable Is Needed: Reverse String

But in recursion,

```
def reverse(s):
    if len(s) <= 1:
        return s
    return reverse(s[1:]) + s[0]
```

- You don’t need a separate variable because each recursive call “remembers” its own partial work in the call stack.
- Each call waits for the smaller version (reverse(s[1:])) to finish before adding its own letter. So, the call stack itself acts as the “storage” for partial results.

Thinking Recursively: Reverse String



Thinking Recursively: Reverse String in Python Tutor

Let’s work together: is Palindrome

A palindrome is a word that reads the same backward and forward (like radar, level, madam).

We want to check if a word is a palindrome using recursion.

Mathematical definition:

- is_palindrome(s)= True, if len <= 1

  False, if s[0] != s[-1]

  is_palindrome(s[1:-1]), otherwise

Let’s work together: is Palindrome

For our problem:

Function name is going to be called is_palindrome(). The input is a string and output is a boolean

Examples:

is_palindrome(“hello”) -> false

is_palindrome(“radar”) -> true

is_palindrome(“level”) -> true

is_palindrome(“python”) -> false

is_palindrome(“a”) -> true

Reminder: is Palindrome Using Loops

```
def is_palindrome(s):
    left = 0
    right = len(s) - 1

    while left < right:
        if s[left] != s[right]:
            return False
        left += 1
        right -= 1

    return True
}
```

Thinking Recursively: is Palindrome

Can I define this problem in terms of itself?

is_palindrome(“radar”) -> s[0] == s[-1] -> ‘r’ == ‘r’?

is_palindrome(“ada”) -> s[0] == s[-1] -> ‘a’ == ‘a’?

Base case! ➡️ is_palindrome(“d”) -> true

Thinking Recursively: is Palindrome

Can I define this problem in terms of itself?

is_palindrome(“radar”) -> s[0] == s[-1] -> ‘r’ == ‘r’?

is_palindrome(“ada”) -> true

Thinking Recursively: is Palindrome

Can I define this problem in terms of itself?

is_palindrome(“radar”) ->true

We got to result! 🥳 But how?🤔

Thinking Recursively: is Palindrome

- Instead of starting from the ends and meeting in the middle like we did iteratively (using a loop), we started from the full word and shrunk it on each recursive call.
- Our base case from what we saw and know about palindromes is that a word with 0 or 1 character is always a palindrome — that's when recursion stops.
- The recursive case is comparing the first and last character of the string. If they're equal, we call the function again on the substring in between (s[1:-1]). If they're different, we return False immediately.

```
def is_palindrome(s):
    # Base case: 0 or 1 characters → always a palindrome
    if len(s) <= 1:
        return True
    # Compare first and last letters
    elif s[0] != s[-1]:
        return False
    else:
        # Recursive case: check the substring excluding first and last chars
        return is_palindrome(s[1:-1])
```

Thinking Recursively: is Palindrome

```python
def is_palindrome(s):
    # Base case: 0 or 1 characters → always a palindrome
    if len(s) <= 1:
        return True
    # Compare first and last letters
    elif s[0] != s[-1]:
        return False
    else:
        # Recursive case: check the substring excluding first and last chars
        return is_palindrome(s[1:-1])
```

⬅️Base case

⬅️early stop

⬆️Recursive case

Thinking Recursively: Palindrome in Python Tutor

Let’s work together: Range Sum

You have a list of numbers and want to find the sum of the numbers between two indexes (start and end) using recursion.

definition:

- range_sum(list,start,end) : 0, if start > end

  list[start]+range_sum(list, start+1, end), otherwise

Example:

range_sum([1, 2, 3, 4, 5], 0, 4) -> 15

1+2+3+4+5 = 15

Let’s work together: Range Sum

For our problem:

Function name is going to be called range_sum(). The input is a list, and two integers and output is a integer

Examples:

range_sum([1, 2, 3, 4, 5], 1, 3) -> 9

range_sum([1, 2, 3, 4, 5], 0, 4) -> 15

Reminder: Range Sum Using Loops

```
def range_sum_loop(lst, start, end):
    total = 0
    for i in range(start, end + 1):
        if i < len(lst):     # Avoid out-of-bounds errors
            total += lst[i]
    return total
```

Thinking Recursively: Range Sum

Can I define this problem in terms of itself?

range_sum([1, 2, 3, 4], 0, 3) -> 1 + range_sum([1, 2, 3, 4], 1, 3)

range_sum([1, 2, 3, 4], 1, 3) -> 2 + range_sum([1, 2, 3, 4], 2, 3)

range_sum([1, 2, 3, 4], 2, 3) -> 3 + range_sum([1, 2, 3, 4], 3, 3)

range_sum([1, 2, 3, 4], 3, 3) -> 4 + range_sum([1, 2, 3, 4], 4, 3)

Base case! ➡️ range_sum([1, 2, 3, 4], 4, 3) -> 0

Thinking Recursively: Range Sum

Can I define this problem in terms of itself?

range_sum([1, 2, 3, 4], 0, 3) -> 1 + range_sum([1, 2, 3, 4], 1, 3)

range_sum([1, 2, 3, 4], 1, 3) -> 2 + range_sum([1, 2, 3, 4], 2, 3)

range_sum([1, 2, 3, 4], 2, 3) -> 3 + range_sum([1, 2, 3, 4], 3, 3)

range_sum([1, 2, 3, 4], 3, 3) -> 4 + range_sum([1, 2, 3, 4], 4, 3)

range_sum([1, 2, 3, 4], 4, 3) -> 0

Thinking Recursively: Range Sum

Can I define this problem in terms of itself?

range_sum([1, 2, 3, 4], 0, 3) -> 1 + range_sum([1, 2, 3, 4], 1, 3)

range_sum([1, 2, 3, 4], 1, 3) -> 2 + range_sum([1, 2, 3, 4], 2, 3)

range_sum([1, 2, 3, 4], 2, 3) -> 3 + range_sum([1, 2, 3, 4], 3, 3)

range_sum([1, 2, 3, 4], 3, 3) -> 4 + 0 ->4

Thinking Recursively: Range Sum

Can I define this problem in terms of itself?

range_sum([1, 2, 3, 4], 0, 3) -> 1 + range_sum([1, 2, 3, 4], 1, 3)

range_sum([1, 2, 3, 4], 1, 3) -> 2 + range_sum([1, 2, 3, 4], 2, 3)

range_sum([1, 2, 3, 4], 2, 3) -> 3 + 4 -> 7

Thinking Recursively: Range Sum

Can I define this problem in terms of itself?

range_sum([1, 2, 3, 4], 0, 3) -> 1 + range_sum([1, 2, 3, 4], 1, 3)

range_sum([1, 2, 3, 4], 1, 3) -> 2 + 7 -> 9

Thinking Recursively: Range Sum

Can I define this problem in terms of itself?

range_sum([1, 2, 3, 4], 0, 3) -> 1 + 9 -> 10

We got to result! 🥳 But how?🤔

Thinking Recursively: Range Sum

- Instead of adding one element at a time inside a loop (moving index by index), we started from the entire range and let each recursive call handle the next smaller part of the list.
- Our base case, from what we saw and know about recursion, is when the starting index goes past the ending index — at that point, there are no more elements left to add, so the function simply returns 0.
- The recursive case is taking the current element at position start, and adding it to the result of calling the same function again on the rest of the range (start + 1).
- Each recursive call works on a smaller version of the problem, moving closer to the base case until all elements have been processed.

```
def range_sum(lst, start, end):
    if start > end:                    # Base case
        return 0
    else:                              # Recursive case
        return lst[start] + range_sum(lst, start + 1, end)
```

Thinking Recursively: Range Sum

```
def range_sum(lst, start, end):
    if start > end:
        return 0
    else:                                    # Recursive case
        return lst[start] + range_sum(lst, start + 1, end)
```

⬅️Base case

⬆️Recursive case

Thinking Recursively: Range Sum in Python Tutor

Let’s work together: Permutations

A permutation is a possible arrangement of all the elements in a list (or string).

Example, for the list [1,2,3] the permutations are:

[1,2,3]

[1,3,2]

[2,1,3]

[2,3,1]

[3,1,2]

[3,2,1]

Let’s work together: Permutations

For our problem:

Function name is going to be called permutations(). The input is a list, and output is lists

Examples:

permutations([1,2,3]) ->

[1,2,3]

[1,3,2]

[2,1,3]

[2,3,1]

[3,1,2]

[3,2,1]

Permutations Using Loops

```
nums = [1, 2, 3]
for i in range(3):
    for j in range(3):
        for k in range(3):
            if i != j and j != k and i != k:
                print(nums[i], nums[j], nums[k])
```

For this problem, you can use nested loops if you know the size

Permutations Using Loops

What is instead of a list with length of 3, it will be 4 or 5?

```
for a in range(4):
    for b in range(4):
        for c in range(4):
            for d in range(4):
                if len({a, b, c, d}) == 4:
                    ...
```

This becomes complicated to maintain… 😅

Thinking Recursively: Permutations

Can I define this problem in terms of itself?

permutations([1, 2, 3])

permutations([1, 2, 3]) -> 1 + permutations([2,3])

permutations([2, 3]) -> 2 + permutations([3])

permutations([3]) -> [3]

Base case!

permutations([1, 2, 3]) -> 1 + permutations([2,3])

⬆️

permutations([2, 3]) -> 3 + permutations([2])

permutations([2]) -> [2] ⬅️Base case!

permutations([1, 2, 3]) -> 1 + permutations([2,3])

permutations([2, 3]) -> [[2 ,3], [3,2]]

permutations([1, 2, 3]) -> 1 + permutations([2,3]) -> [1,2,3], [1,3,2]

permutations([1, 2, 3]) -> 2 + permutations([1,3])

⬆️

permutations([1, 3]) -> 1 + permutations([3])

permutations([3]) -> [3] ⬅️Base case!

permutations([1, 2, 3]) -> 2 + permutations([1,3])

⬆️

permutations([1, 3]) -> 3 + permutations([1])

permutations([1]) -> [1] ⬅️Base case!

permutations([1, 2, 3]) -> 2 + permutations([1,3])

⬆️

permutations([1, 3]) -> [[1 ,3], [3,1]]

permutations([1, 2, 3]) -> 2 + permutations([1,3]) -> [2,1,3], [2,3,1]

permutations([1, 2, 3]) -> 3 + permutations([1,2])

⬆️

permutations([1, 2]) -> 1 + permutations([2])

permutations([2]) -> [2] ⬅️Base case!

permutations([1, 2, 3]) -> 3 + permutations([1,2])

permutations([1, 2]) -> 2 + permutations([1])

permutations([1]) -> [1] ⬅️Base case!

permutations([1, 2, 3]) -> 3 + permutations([1,2])

permutations([1, 2]) -> [[1 ,2], [2,1]]

permutations([1, 2, 3]) -> 3 + permutations([1,2]) -> [3,1,2],[3,2,1]

Thinking Recursively: Permutations

Can I define this problem in terms of itself?

permutations([1, 2, 3]) ⟶

[[1, 2, 3], [1, 3, 2],

[2, 1, 3], [2, 3, 1],

[3, 1, 2], [3, 2, 1]]

We got to result! 🥳 But how?🤔

Thinking Recursively: Permutations

- Instead of using nested loops to place each element manually (one loop per position), we started from the entire list and let each recursive call handle the next smaller sublist of remaining elements.
- Our base case, from what we know about recursion, is when the list has only one element left — at that point, there's only one possible arrangement, so the function simply returns that list inside another list (e.g., [[3]]).
- The recursive case is selecting one element to fix in the front, and then calling the same function again on the remaining elements to find all their permutations.
- Each recursive call works on a smaller version of the problem, gradually building all combinations until every possible ordering of the original list has been generated

```python
def permutations(nums):
    # Base case #1: nums is empty returns empty list (no permutations)
    if len(nums) == 0:
        return []
    # Base case #2: nums only has one number returns a list with only one permutation
    # (for example, [1] has only one number so the only perm is [1], so returns [[1]].
    if len(nums) == 1:
        return [nums]
    # If not a base case, iterate through each number in nums, get the permutations of the remaining
numbers in the list
    # and add the original number alongside all the returned permutations as a new permutation.
    #
    # For example, if the list is [1, 2, 3], begin by saving 1. Get the permutations of the remaining
numbers, which
    # are [2, 3] (this returns a list with [2, 3] and [3, 2]). Add the 1 at the start (or end) of [2,
3] and [3, 2]
    # and that creates the new permutations [1, 2, 3] and [1, 3, 2]. Continue the process by saving
number 2,
    # generating the permutations of [1, 3], and repeating the process.
    perms = []
    for i in range(len(nums)):
        curr_num = nums[i]
        remaining_nums = nums[:i] + nums[i+1:]
        for perm in permutations(remaining_nums):
            new_perm = [curr_num] + perm
            perms.append(new_perm)
    return perms

print(permutations([1, 2, 3]))
```

Thinking Recursively: Permutations in Python Tutor

Questions?

Post to our Discord server

or

email:

upr-ciic3015-staff@upr.edu

karelys.lopez1@upr.edu

Give us anonymous feedback!

https://bit.ly/CIIC3015_F25_Feedback

CODE PATH .ORG